LONDON'S GLOBAL UNIVERSITY

UCL

Model-Agnostic Meta-Learning for Few-Shot Deep Learning

Josh Strong

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Master of Science
of
University College London.

Department of Statistical Science
University College London

Word Count: 9,760
September 14, 2020

I, Josh Strong, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

Learning to learn is a phrase commonly used to describe meta-learning, a subfield of machine learning which equips models with prior information, enabling them to be able to better tackle few-shot learning problems, in which very few amounts of data are available to train models. In this project, I begin with a short literature review on the topic of deep learning and meta-learning, in the context of solving few-shot learning problems. This knowledge is then used to explore a popular optimisation-based meta-learning algorithm, Model-Agnostic Meta-Learning (MAML). MAML is detailed, along with proposed variants of this algorithm; MAML++, NIL and ANIL, which seek to improve on the potential weaknesses of MAML. The project is concluded with two experimental studies, in which the findings of this report are applied. The first and primary experimental study conducted involved a modified version of a toy-problem first demonstrated in the original MAML article, where the effectiveness of MAML is tested using an extra level of difficulty. The secondary study applies MAML to a computer-vision related task, involving the Fewshot-CIFAR100 dataset. MAML is shown to be a simple yet effective tool for tackling few-shot learning problems, but yet still comes with its own drawbacks and difficulties.

Acknowledgements

Many thanks to my supervisors, Dr Jinghao Xue and Miss Xiaochen Yang, for their patience and guidance throughout this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Aim of the Project

1.1.1 Few-Shot Learning

Artificial intelligence has made promising progress in recent times due to advances in hardware, predominantly GPUs and TPUs, which can take use of large data sets to quickly train sophisticated models.

AlphaGO, a computer program developed by a team at DeepMind, shocked the world when it became the first computer program to defeat the GO world champion and 9 dan ranked[1] Lee Sedol, in March 2016 [11]. Subsequently, AlphaGO has become regarded by many as the strongest player in the game's history. The artificial intelligence architecture behind AlphaGo consisted of *value* and *policy* deep neural networks, trained by a mixture of supervised and reinforcement learning techniques [12]. Self-driving cars, a form of artificial intelligence recently thought as being a staple part of a technologically advanced and distant-future civilisation, has made leaps and bounds towards autonomy through the use of convolutional neural networks;

[1] 9 dan rank refers to the highest ranking achievable in GO, a feat reached by fewer than the top 200 players world-wide.

a sophisticated architecture form of neural networks designed specifically for computer vision tasks. Tesla's Autopilot is a modern-day realisation of this triumph where, with sufficient funds, anyone can buy a car with *"advanced hardware capable of providing Autopilot features today, and full self-driving capabilities in the future - through software updates designed to improve functionality over time"* [13].

Unfortunately, the advancement of artificial intelligence has also brought into question how ethical these programs are. In 2018, a huge scandal erupted in the UK and US in what has become known as the Facebook-Cambridge Analytica data scandal, where millions of unconsenting Facebook users unwittingly had their private data extracted and used for political advertising [14]. The purpose of this mass data abduction was thought to be used for artificial intelligence models, sparking calls for regulation in the use of such ever-increasingly powerful and data-hungry programs [15].

Although such artificial intelligence models have came a long way since the days of when the perceptron algorithm was first invented by Frank Rosenblatt in 1958 [16], the main problem at hand in modern-day time is that such deep neural network architectures do not generalise[2] well once trained on small-samples of data. Typically, upwards of thousands of parameters are used in order to capture the underlying true complex nature of the relationship between input and output. Without a sufficiently large data set to compensate for these large amounts of parameters, this often leads to overfitting; a phenomenon which will be described in more detail in the following subsection. A common example used in introductory few-shot learning literature sections, to provide a concrete example of the pitfalls of artificial

[2]The generalisation of a model refers to how well it performs to data it has not trained on, and has yet to see.

intelligence models in comparison to human intelligence, is that of children learning animal names; within seeing animals only a few times, children can subsequently identify additional animals from henceforth. This is certainly not the case for deep neural networks, and a feature greatly desired in the progression of artificial intelligence towards human intelligence. This issue gave rise to Few-Shot Learning (FSL), an umbrella term given to machine learning techniques used to tackle these problems of small-samples of data used in conjunction with deep learning. With the possibilities of massively reducing the time-consuming burdens of data-collection and training models, few-shot learning is indeed a crucial subject area of machine-learning, required in the advancement of artificial intelligence.

1.1.2 Meta-Learning in Deep Learning

Traditionally, when training neural networks for machine learning tasks, a network is trained to learn a specific task[3] at hand as optimally as possible, such that it generalises well on unseen test data. The process of which the model learns begins from scratch; a model is initialised with random parameters with no prior information given as aid. As a result, training is often data intensive, computationally expensive and time consuming. In meta learning, we are *learning to learn*. A network is trained to gain experience and knowledge about a distribution of tasks by training on a variety of tasks sampled from this distribution, posed as a form of prior information. Through this background information, meta-learned neural networks do not need masses of data, as they usually do, in order to generalise to new tasks. The models are readily equipped with experience allowing them to quickly generalise to new, unseen tasks with few amounts of data. This makes meta learning an

[3]A task, in this setting, can be interpreted as a specific machine learning problem. Further details are given later in the report, in Section 2.2.

excellent solution to few-shot learning problems.

The potential benefits meta-learning holds in deep learning has spurred an elation in the quantity of research in the topic, across many subject areas.

In this project, I investigate a specific algorithm residing in a subfield of meta-learning, optimisation-based meta-learning, named Model-Agnostic Meta-Learning.

1.1.3 Model-Agnostic Meta-Learning

Model-Agnostic Meta-Learning (MAML) [6] is a versatile optimisation-based meta-learning algorithm suited for few-shot learning problems. The algorithm is named "Model-Agnostic", such that it can be applied to any model which is trained using gradient descent[4], but an emphasis is placed on deep learning models in this project which, as previously described, suffer from poor performance when trained few amounts of data. In short, MAML works by using prior knowledge, in the form of a set tasks, to learn *meta-parameters* for a model. These parameters readily facilitate fast adaptation to novel tasks with few amounts of data, such that good generalisation is achieved and overfitting is avoided.

1.1.4 Aim and Accomplishments of the Project

The primary aim of this project is to conduct research into how MAML can be used to tackle few-shot learning problems. This involves studying MAML, including its variants, strengths and weaknesses, as well as additional literature scrutinising the algorithm, all with a focus with using deep learning. This is done by first establishing a brief background into deep learning and meta-learning, then by reviewing recent research through academic articles and studies, before concluding with two experimental studies

[4]Gradient descent is a primitive method through which models are *trained*, such that their objective performance increases. Further details are given in Section 2.1.3.

to further develop understanding onto described methods. In conducting the experimental studies, two notebooks written in Python, which implement different methods explored in this project.

1.2 Report Structure

The report consists of 5 main sections; an introduction, a literature review, an overview of the methods used in this report, experimental studies, and a final section describing the closing conclusions of the project. The literature review entails briefly exploring theory of deep learning, contingent for subsequent material in this project, and introducing meta-learning in the context of solving FSL problems. The methods section provides the fine details of MAML itself, details a variant of MAML named MAML++, and explores an article which evaluates the effectiveness of MAML in addition to proposing modified algorithms named ANIL and NIL. The following experimental studies section formalises the two experimental studies conducted amongst reporting findings. The final section on conclusions delves into the general conclusions of the project, including limitations and potential future work.

Chapter 2

Literature Review

2.1 A Brief Introduction to Deep Learning

Deep learning encapsulates a set of models, consisting of artificial neural networks, for which this project makes great use of. This section details a brief introduction to the necessary content of deep learning required for the following content of the project. Beginning with the elementary components of which these networks are built upon, perceptrons, we then build up to multilayer perceptrons; the most commonly-recognised structure of neural networks. The nature of how these models are efficiently and effectively trained, backpropogation, is then described. The section ends by studying convolutional neural networks, a further-developed form of these neural networks, which achieve great performance on computer-vision related tasks through its highly-specialised architecture.

2.1.1 Perceptrons

The following material on perceptrons and feedforward neural networks is based upon the work of Goodfellow et al., in their book "Deep Learning" [17]. Before getting into the detail of neural networks, it is important to break down the infrastructure behind the scenes. Neural networks, in their

foundations, are made up of many perceptrons.

The perceptron is a primitive algorithm, considered by many as the first and simplest neural network. First introduced in 1958 by F. Rosenblatt in his paper "*The Perceptron: A Probabilistic Model For Information Storage and Organization in the Brain*" [16], and later refined by Minsky and Papert [18], the perceptron is used as a binary classification model for supervised learning problems, such that given a set of labelled data belonging to one of two classes, it can form a prediction for which class this data belongs to. The main assumption for the perceptron algorithm to work, is that the data is *linearly separable*[1].

Consider a simple example, where we have a set of N data points $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$ where $\boldsymbol{x}_i$ denotes the three inputs for the i^{th} data point, $\boldsymbol{x_i} = (1, x_{i,1}, x_{i,2}, x_{i,3})$ and $y \in \{0, 1\}$. Note that we have included an intercept term for each data point. Each input, including the intercept term, is assigned a weight $\boldsymbol{w} = (w_0, w_1, w_2, w_3)$. These weights all form the parameters of the model, which are usually randomly initialised to small values close to or exactly 0. The perceptron makes a classification judgement for a data point, based on the linear combination of its inputs and their respective weights. We denote the linear predictor for the i^{th} data-point, as η_i:

$$\eta_i = w_0 + \sum_{j=1}^{3} w_j x_{i,j},$$

where $x_{i,j}$ denotes the j^{th} input for the i^{th} data point and w_j denotes the weight for the j^{th} input. An activation function f, in the form of the Heaviside step function, is then applied to this linear combination, which results

[1]Linearly separable data implies that if the data were plotted on a graph, then there exists a separating hyper-plane which separates the two sets of data.

in an activation equivalent to the prediction $\hat{y}_i$ for this data point, $f(\boldsymbol{x}_i) = \hat{y}_i$

$$f(\boldsymbol{x}_i) = \begin{cases} 1, & \text{if } \eta_i > 0 \\ 0, & \text{otherwise.} \end{cases}$$

Since the parameters for the perceptron are randomly initialised, the predictions are not usually accurate. In order for this model to generate accurate predictions, it needs to learn goods values for parameters $\boldsymbol{w}$. The process by which the model does this is through the perceptron learning algorithm, as detailed in Algorithm 1.

Algorithm 1: The Perceptron Learning Algorithm.

Result: Learn good parameters to accurately classify using training dataset $\mathcal{D}$.

Initialise $\boldsymbol{w}$ randomly to values close to or exactly 0;

Choose learning rate α;

for $i = \{1, ..., N\}$ **do**

(i) Calculate perceptron prediction: $\hat{y}_i = f(\boldsymbol{x}_i)$;

(i) Update weights: $\boldsymbol{w} = \boldsymbol{w} + \alpha(y_i - \hat{y}_i)\boldsymbol{x}_i$

end

The learning algorithm relies on a learning rate α, a constant which determines the extent to which the model alters the weights. A learning rate is a *hyperparameter* of the model, a common feature of machine-learning algorithms. The process by which the Perceptron "learns" fits into the general structure as to how all machine learning algorithms learn. This, along with learning rates, are further detailed in the next subsection. After training, the learned parameter weights results in greater classification accuracy for the Perceptron. Figure 2.1 visually displays the aforementioned structure of

the perceptron, which is useful to keep in mind when considering multilayer perceptrons.

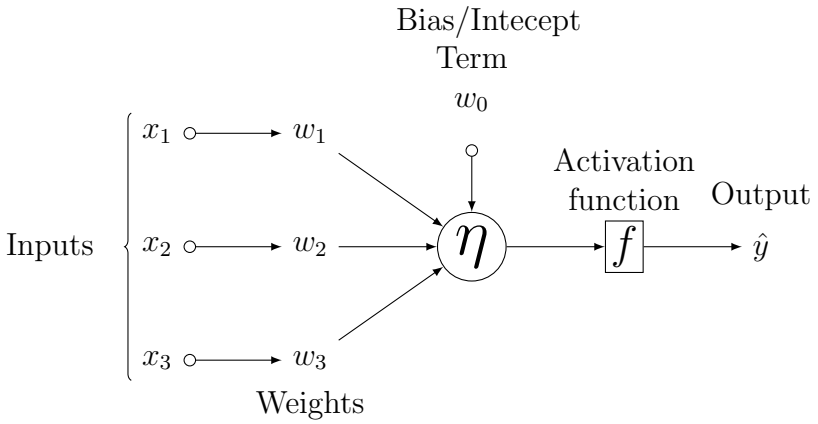


Figure 2.1: A perceptron. *Source: Modified code from Medina* [1].

2.1.2 Multilayer Perceptrons and Feedforward Neural Networks

More often than not, the perceptron is not sophisticated enough of a model to capture the often complex and non-linear underlying true relationship between input and output, for real-world problems. However, the toolkit a perceptron provides can be extended into a multitude of models which can capture such relationships, by introducing stacked layers of multiple perceptrons. Here, perceptrons are equivalently denoted as *neurons*, and we begin by defining notation used for multilayers of perceptrons.

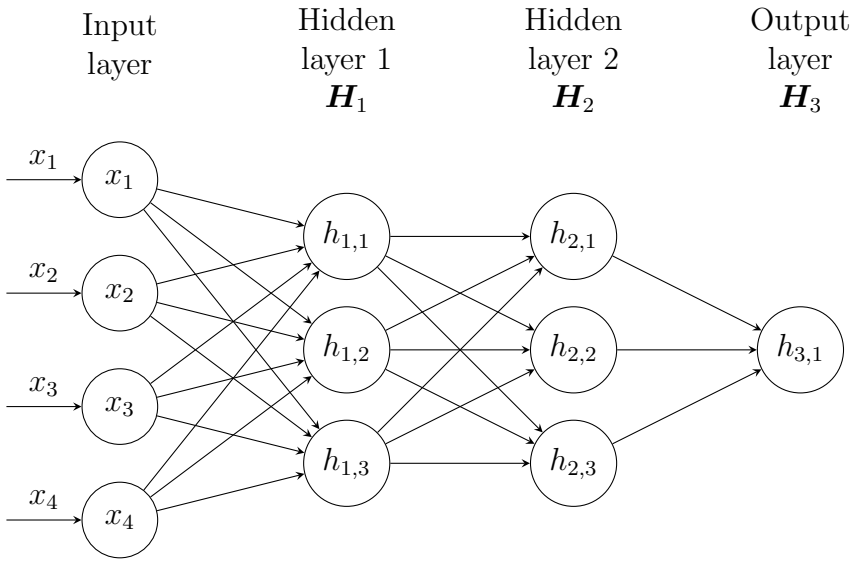


Figure 2.2: A graph depicting the structure of a neural network with 4 inputs, 2 hidden layers, each with 3 neurons, and 1 output.

Given K_i neurons in the i^{th} layer[2] of a network, we denote the activation of the j^{th} neuron in this layer by $h_{i,j}$. For future purposes, we now recursively define an entire layer of activations of the network for an arbitrary ℓ^{th} hidden layer. All activations of the first hidden layer of the network is given by

$$\boldsymbol{H}_1(\boldsymbol{x}) = f(\boldsymbol{W}_1^T \boldsymbol{x} + \boldsymbol{b}_1) \in \mathbb{R}^{K_1},$$

where

- $W_1 = [\boldsymbol{w}_1, ..., \boldsymbol{w}_{K_1}] \in \mathbb{R}^{N \times K_1}$ is a matrix of weights, connecting the neurons in the input layer to the first hidden layer,
- $\boldsymbol{b}_1 = [b_1, ..., b_{K_1}]^T \in \mathbb{R}^{K_1}$ is a vector of biases used for the first hidden layer.
- and the activation function $f(\cdot)$ is applied element-wise.

The activations of the second layer of the network can be expressed as a

[2]Not including the input layer.

function of the activations of the first layer of the network:

$$\boldsymbol{H}_2(\boldsymbol{H}_1(\boldsymbol{x})) = f(\boldsymbol{W}_2^T \boldsymbol{H}_1(\boldsymbol{x}) + \boldsymbol{b}_2) \in \mathbb{R}^{K_2},$$

where

- $\boldsymbol{W}_2 \in \mathbb{R}^{K_1 \times K_2}$ is a matrix of weights, connecting the neurons in the first hidden layer to the second hidden layer,
- $\boldsymbol{b}_2 \in \mathbb{R}^{K_2}$ is a vector of biases used for the second hidden layer.

Through repeat substitution, we find that the ℓ^{th} hidden layer of activations can be written as

$$\boldsymbol{H}_\ell(\boldsymbol{H}_{\ell-1}(...(\boldsymbol{H}_1(\boldsymbol{x}))) = f(\boldsymbol{W}_\ell^T \boldsymbol{H}_{\ell-1}(...(\boldsymbol{H}_1(\boldsymbol{x})) + \boldsymbol{b}_\ell) \in \mathbb{R}^{K_\ell},$$

where $\boldsymbol{W}_\ell^T \in \mathbb{R}^{K_{\ell-1} \times K_\ell}$ and $\boldsymbol{b}_\ell \in \mathbb{R}^{K_\ell}$. From this, it can be seen mathematically that the activations in an arbitrary ℓ^{th} layer are a function of activations in all preceding layers. In this sense, information, in the form of activations in a neural network, is said to pass forward through the network, from the input to the output. There are no loops of this type of network, such that information passed forward cannot reach previous stages in the network, hence multilayer perceptrons are under the category of *feedforward* neural networks.

There exists many activation functions which are used in multilayered perceptrons. Sigmoid, ReLU and Tanh are some commonly used activation functions, which introduce non-linearity to the network. Non-linearity in a crucial feature of the network, as without it the network would simply ouput an affine transformation of inputs and be unable to capture any complex underlying relationships between the inputs and output. This is demonstrated in Appendix A.1.

2.1.3 Backpropogation and Autodiff Libraries

Now that we have the framework of a model capable of learning complex relationships, we now describe the process for which neural networks are trained; *backpropogation*.

An untrained neural network outputs a prediction $\hat{y} = f(\boldsymbol{x}; \boldsymbol{w})$ for a set of training data $\mathcal{D} = \{(\boldsymbol{x}, y)\}_{i=1}^{N}$. A loss function[3], $\mathcal{L}(y, \hat{y})$, can be used in conjunction with the true label y and the network output to quantify the relative margin of error of the model, in its current state, for this set of data. In training the model to improve its performance across training data, we can iteratively adjust the networks parameters, previously described as weights, such that the parameters of the network are moved in the direction which will incur the steepest descent of the loss, with respects to each and every parameter. This primitive technique, known as *gradient descent*, is a common first-order optimisation algorithm applied in machine learning problems to find local minima of differentiable loss functions, favoured for its efficiency, simplicity, and extensions. Gradient descent works as follows [19]:

Given a differentiable function $J(\boldsymbol{w})$, start with some initialisation of $\boldsymbol{w}$ and repeatedly update until convergence:

$$w_j := w_j - \alpha \frac{\partial}{\partial w_j} J(\boldsymbol{w}), \tag{2.1}$$

for all parameters w_j simultaneously, where $\alpha \in \mathbb{R}$ is the learning rate of the algorithm, a hyperparameter which determines the extent to which the

[3]The loss function used in a neural network depends upon the task at hand, but most typically the loss function for classification problems involves cross entropy, and mean squared error for regression.

parameters are updated.

There exists many sophisticated variants and extensions of the gradient descent algorithm above, under the category of gradient-based optimisation techniques, one of which being relevant to this project; Adam [20]. Adam works similarly to gradient descent, such that it is a first-order optimisation algorithm which iteratively updates parameters. The authors of Adam describe it as computationally efficient, straightforward to implement, requiring little memory in addition to having many other beneficial features. Unlike gradient descent, Adam does not have a constant learning rate fixed for all parameters, but instead a dynamic learning rate is individually assigned and learned for each parameter of the model as training progresses. The empirical results of Adam, as shown in the original paper, using several different model architectures to classify handwritten digits 0-9 in the MNIST database [21], displays the effectiveness of Adam in optimisation: *"Using large models and datasets, we demonstrate Adam can efficiently solve practical deep learning problems"*.

The basis for which these gradient-based optimisation techniques work is centered around partial derivatives of a differentiable loss function with respects to parameters, as seen in the gradient descent algorithm in equation 2.1. There exists a few methods for determining these gradients such as numerical differentiation, symbolic differentiation or manual differentiation. One family of methods, known as automatic differentiation, or colloquially as autodiff, has became the prominent method used in modern day machine learning due to its efficiency and accuracy in calculating derivatives. The article *Automatic Differentiation in Machine Learning: a Survey* by Bay-

din et al. provides an excellent explanation and comparison of autodiff to alternative methods [22]. In short, autodiff views multivariable differentiable functions as being comprised of a finite set of elementary arithmetic operations[4] and elementary functions[5], for which the derivatives of such operations and functions are known. The derivative of the overall composition of these variables can be derived through combining derivatives of the individual operations, using the chain rule. Such compositions can be represented by computation graphs with input, intermediary and output variables being represented by nodes. An example computation graph of the function $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ can be seen in Figure 2.3.

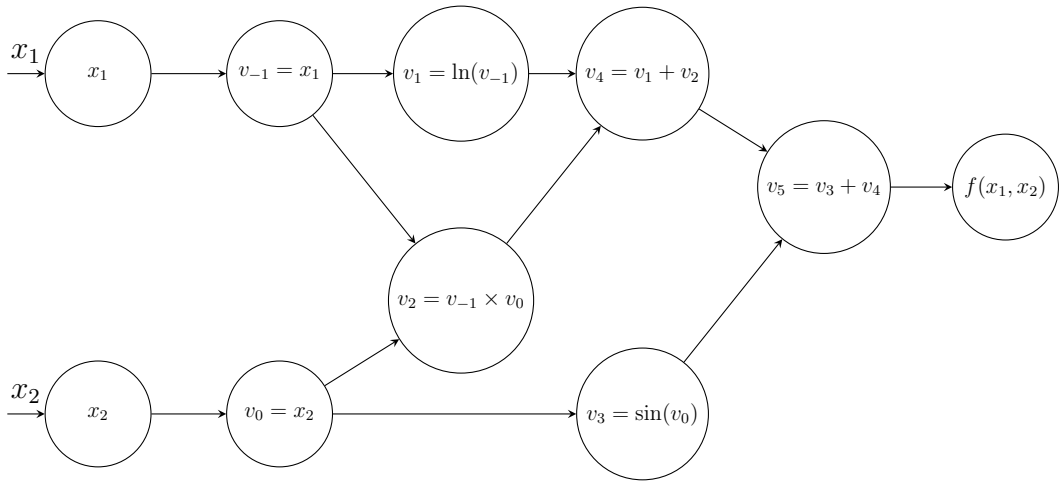


Figure 2.3: The computation graph of the function $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$, using Griewank-Walther notation [2].

Autodiff takes a function $f(\boldsymbol{x})$ and returns an exact value, depending on machine accuracy, for the gradient evaluated at $\boldsymbol{v}$:

$$g_i(\boldsymbol{v}) \equiv \frac{\partial}{\partial x_i} f(\boldsymbol{x})\bigg|_{\boldsymbol{x}=\boldsymbol{v}}, \; i = 1, ..., m.$$

[4]Arithmetic operations being the usual addition, subtraction, multiplication or division.

[5]Elementary functions being functions such as trigonometric functions, polynomial functions, exponentiation functions, etc.

Forward and reverse mode automatic differentiation are the two modes under the umbrella of automatic differentiation for which one can compute these gradients. Forward mode autodiff involves repeatedly applying the chain rule in a forwards-wise fashion to successive computational nodes of a computation graph, arranged in a forward schedule, and propagating derivatives of parent nodes with respects to the input parameter until you can compute the derivative of the desired function with respects to the desired input. The partial derivative of a node v with respects to an input parameter x, denoted $\dot{v}$, is calculated for nodes by propagating gradients of parent nodes[6]:

$$\dot{v} := \frac{\partial v}{\partial x} = \sum_{v_i \in \text{Parent}(v)} \dot{v}_i \frac{\partial v}{\partial v_i}.$$

Reverse mode autodiff is similar to forward mode, except here the computation graph is traversed from outputs to inputs, applying the chain rule in a backwards-wise fashion, propagating gradients of child nodes[7] to parent nodes. The partial derivative of an output y with respects to an intermediate variable v, denoted $\bar{v}$, is

$$\bar{v} := \frac{\partial y}{\partial v} = \sum_{v_i \in \text{Child}(v)} \frac{\partial v_i}{\partial v} \bar{v}_i.$$

For further clarification with an example on autodiff, table B.1 details calculations of both forward and reverse mode automatic differentiation, for the function $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ as seen in the computation graph in Figure 2.3.

Backpropogation is the name given to the use of reverse mode automatic

[6]Parent nodes of a given node are all nodes from which that node extended from.
[7]Child nodes of a given node are all nodes extending from that node.

differentiation in backpropagating losses with respects to parameters in multilayer perceptrons. Reverse mode of automatic differentiation is preferred to forward mode as the method of computing gradients for neural networks because usually in machine learning tasks, we have a huge amount of inputs and relatively smaller amounts of outputs. Reverse-mode autodiff is better suited to this problem, as with a neural network represented by a function $f : \mathbb{R}^n \to \mathbb{R}^m$ with $n \gg m$, only m backwards passes of the computation graph are required, in comparison to n forwards passes required if using forward mode. This can be seen by considering the resulting Jacobin vector products using forward and reverse mode automatic differentiation, which is further detailed in Appendix B.2.

Automatic differentiation had been widely implemented in machine-learning libraries, such as PyTorch [23] and TensorFlow [24] in Python [25] and Flux [26] in Julia [27]. Such libraries have GPU support, enabling faster training with the use of large data sets.

2.1.4 Convolutional Neural Networks

Stacked layers of multiple perceptrons, as seen in the prior subsections, are themselves also the basis for much more powerful and highly specialised neural networks, such as recurrent neural networks, convolutional neural networks and autoencoders, to name a few. This subsection details a particular variant, relevant to this project, of convolutional neural networks (CNNs). The theory in this section is based upon the excellent material given in Stanford's *Convolutional Neural Networks for Visual Recognition* course [28]. For a deeper understanding into CNNs and their application to computer vision tasks, this course is highly recommended to the reader.

Yann LeCun is considered by many as a founding father of convolutional networks. Convolutional neural networks came to prominence through his paper *Gradient-Based Learning Applied to Document Recognition* [3], where he gave groundbreaking empirical results demonstrating the power of a CNN variant in comparison to then-existing alternative methods. The model, named LeNet-5, was trained to recognise hand-written digits from the MNIST database and achieved a sub-1% testing error rate. The architecture of the 7-layered CNN can be seen in Figure 2.4. In more recent times, models such as Residual Network (ResNet) have produced state-of-the-art empirical results on benchmark data sets. Developed by He et al. and detailed in their article *Deep Residual Learning for Image Recognition* [29], ResNet benefits from an allowance in increased depth of the network due to easier optimisation.

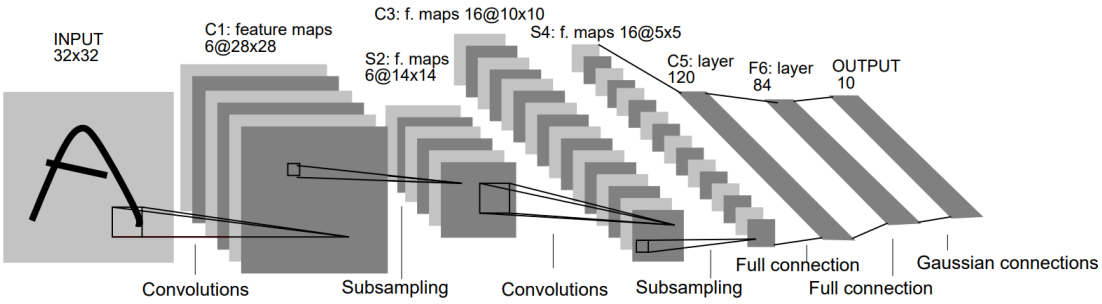


Figure 2.4: Architecture of LeNet-5. *Source: LeCun et al.* [3].

The structure and architecture of the convolutional neural network given in Figure 2.4 displays some of the common layers types. These layers can be broken down into feature learning and classification layers. Feature learning layers, including convolutional and pooling layers, aims to learn useful features such as edges or colours in lower levels of the network and high-level features such as objects in later layers of the network. Classification layers,

similar to multilayer perceptrons, connect the feature learning section of the CNN in order to compute a classification output for the network. These layers are displayed in Figure 2.5, and are further detailed below:

- Feature Learning Layers:
    1. **Convolutional Layers**: Convolutional layers consists of convolving learnable filters, consisting of matrices of weights, across the spatial dimensions of input volumes, by taking dot products of filter weights and filter inputs. The output depends on 3 hyperparamaters: depth, stride and zero-padding.
    2. **Pooling Layers**: Pooling layers serve as a down-sampling operation across the spatial dimensions of the output of convolutional layers, working in a similar fashion to the filters of convolutional layers by convolving across spatial dimensions. One particular type of pooling, max pooling, consists of outputting the maximum number in the pooling filter.
- Classification Layers:
    1. **Fully Connected Layers**: The classification layers are fully-connected, such that each perceptron in a given layer is connected to each perceptrons in adjacent layers. These behave similarly to a multilayer perceptrons.

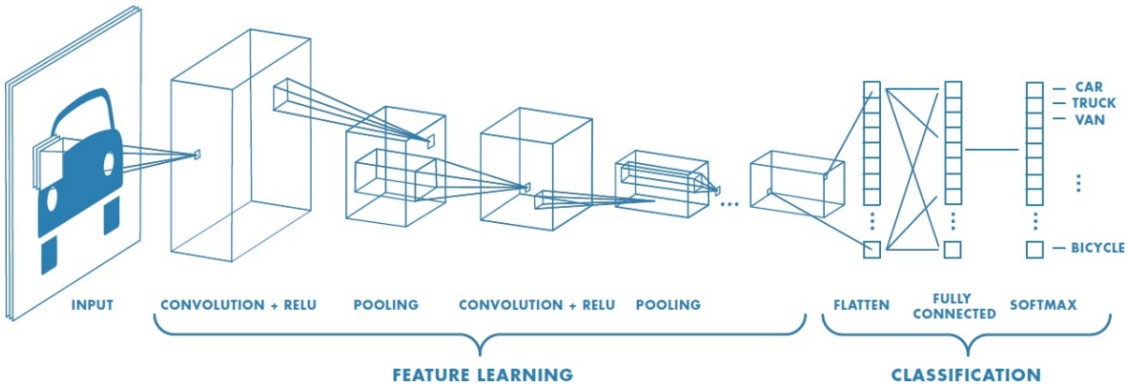


Figure 2.5: Example architecture of a CNN broken down into input, feature learning and classification sections. *Source: MathWorks* [4].

CNNs pose several advantages over multilayer perceptrons in image-recognition tasks:

- Built for images: When considering using an image as an input for a simple multilayer perceptron, where each pixel of the image consists of a single input neuron, it can be seen that the number of parameters can reluctantly and quickly grow out of hand. For the MNIST dataset, the images consist of 28×28 pixels; equating to an input dimension of 784. When factoring in that such networks will typically require numerous and large hidden layers, in addition to the fact that with images we would also like to take use of RGB channels[8], this problem becomes abundantly apparent. CNNs take advantage of the spatial structuring of the data in images.
- Regularisation effect: Since multilayer perceptrons are fully-connected, in addition to the fact that as previously described we face excessive parameters, multilayer perceptrons are often prone to overfitting such that the large number of parameters allow for the model to memorise

[8]RGB channels refer to the red, green and blue colour channels commonly used in images.

training data and generalise poorly to unseen testing data. In comparison to CNNs, which only requires few parameters for every feature map, CNNs do not exhibit this problem and are considered to be a regularised version of multilayer perceptrons.

- Translation invariance: Translation invariance, the concept that no matter how an object is translated it can still be recognised, is built into CNNs through their convolutional and pooling layers of the network. multilayer perceptrons do not have this feature.

2.2 Meta-Learning in Solving Few-Shot Learning Problems

This section begins by detailing the problem framework of the few-shot learning (FSL) problem, in addition to detailing how an unreliable empirical risk minimiser stands as the core issue of FSL. Through modifying the algorithm in optimising deep-learning models, we can remedy the FSL issues, leading on to how refining meta-learned parameters can be used as prior knowledge to more efficiently search in the hypothesis space $\mathcal{H}$ for parameters θ of best hypothesis h^*. We then delve into the fundamentals behind meta-learning, including a brief introduction to meta-learning, and the Bayesian probabilistic perspective of meta-learning, a perspective useful for gaining further understanding on meta-learning.

2.2.1 Few-Shot Learning and its Core Issue

Few-shot learning consists of learning a novel task $\mathcal{T}$ with training data $D^{tr} = \{(x_i, y_i)\}_{i=1}^{I}$ and testing data $D^{test} = \{x^{test}\}$, where I is small. N-way K-shot classification consists of the supervised machine learning problem in which we have N classification categories, each with K data points. In this

case, we have $I = KN$ data points. The main problem with FSL can be broken down and reduced to be a problem due to an unreliable empirical risk minimiser. This can be shown through the decomposition of prediction error in supervised learning problems. This is shown, as follows [5]:

When considering the ground truth between input x and output y being modelled with the joint probability distribution function $p(x, y)$, machine learning problems seek to minimise a loss function $\ell(h(x), y)$ with respects to $p(x, y)$, where h represents a hypothesis from x to y. This can be interpreted as minimising its expected risk, $R(h)$:

$$R(h) = \int \ell(h(x), y)\, dp(x, y) = \mathbb{E}\left[\ell(h(x), y)\right] \tag{2.2}$$

The ground truth $p(x, y)$ is not known, therefore the empirical risk $R_I(h)$ is used instead, as a proxy, in training:

$$R_I(h) = \frac{1}{I} \sum_{i=1}^{I} \ell(h(x_i), y), \tag{2.3}$$

where I denotes the number of training samples. $R_I(h)$ can be considered as as the average loss with hypothesis h.

Considering the two risk functions in equations (2.2) and (2.3), and a hypothesis space $\mathcal{H}$ for h, we have 3 different hypotheses:

1. $\hat{h} = \arg\min_h R(h)$, the function minimising the expected risk (2.2),
2. $h^* = \arg\min_{h \in \mathcal{H}} R(h)$, the function minimising (2.2) within the hypothesis space $\mathcal{H}$,
3. The *empirical risk minimiser*: $h_I = \arg\min_{h \in \mathcal{H}} R_I(h)$, the function minimising the empirical risk (2.3) in $\mathcal{H}$,

where $\hat{h}, h^*$ and h_I are assumed to be unique. The total error can then be

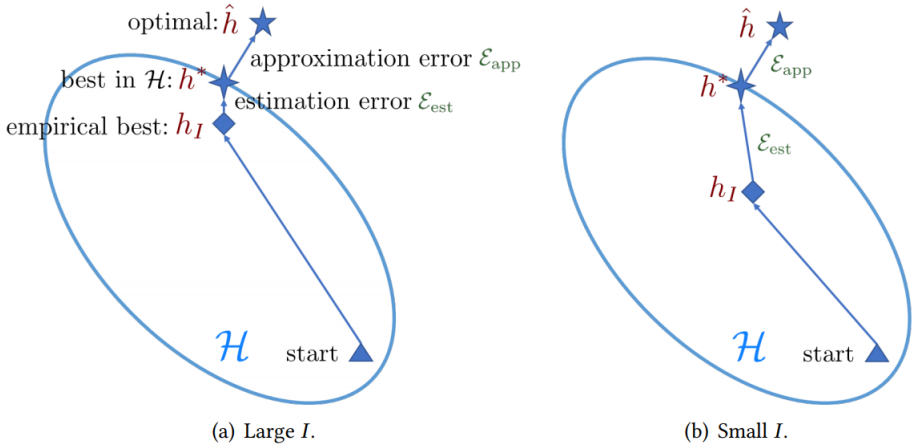
split up in terms of these different hypotheses:

$$\mathbb{E}[R(h_I) - R(\hat{h})] = \underbrace{\mathbb{E}[R(h^*) - R(\hat{h})]}_{\mathcal{E}_{\text{app}}(\mathcal{H})} + \underbrace{\mathbb{E}[R(h_I) - R(h^*)]}_{\mathcal{E}_{\text{est}}(\mathcal{H},I)},$$

where $\mathcal{E}_{\text{app}}(\mathcal{H})$ measures how well functions in the $\mathcal{H}$ can approximate $\hat{h}$ and $\mathcal{E}_{\text{est}}(\mathcal{H}, I)$ measures how well $R_I(h)$ approximates $R(h)$ within $\mathcal{H}$.
From this it can be seen that the total error can therefore be reduced through altering the hypothesis space $\mathcal{H}$ and improving the number of samples I. Wang et al. [5] proposed that this can be done through 3 methods:

1. **Data**: Augment training data to increase training samples I to $\tilde{I}$ with $\tilde{I} \gg I$, resulting in a much better $R_I(h)$ approximation to $R(h)$ and hence better optimal h_I.
2. **Model**: Constrain the complexity of $\mathcal{H}$ to smaller subspace $\tilde{\mathcal{H}}$, resulting in more reliable h_I.
3. **Algorithm**: Use prior knowledge for a better initialisation of the algorithm, or better searching procedure.

In the context of FSL, the main problem in reducing the total error lies within $\mathcal{E}_{\text{est}}(\mathcal{H}, I)$. Since we have only a small number of training points I, $R_I(h)$ is a bad approximation of $R(h)$ hence the optimal h_I is said to overfit, leading to an unreliable empirical risk minimiser. This problem of having small number of data points I can be best understood visually, for which a diagram in Figure 2.6 contrasts the two scenarios.



(a) Large I. (b) Small I.

Figure 2.6: Comparison of few and many data points I affecting $\mathcal{E}_{\text{est}}(\mathcal{H}, I)$. *Source: Wang et al.* [5].

The techniques investigated in this project lie within the algorithm category above. Modifying the algorithm of which optimal parameters θ of best hypothesis h^* are searched for in a hypothesis space $\mathcal{H}$ involves taking advantage of prior knowledge in the form of a meta-learner or learned initialisations for the model parameters, θ_0. This can be done through 3 methods; learning the optimizer, refining existing parameters and refining meta-learned parameters. One such of method of modifying the algorithm consists of refining meta-learned parameters, for which we find Model-Agnostic Meta-Learning.

2.2.2 Introduction to Meta-Learning

The key idea behind meta-learning is that by learning prior knowledge in the form of across-task information, models can then quickly adapt to new tasks such that task-specific knowledge allows for good generalisation performance using only a small amount of data. Few-shot meta-learning is said to have two-stages [30]:

1. Meta-learner component: Prior or shared knowledge is learned from the base data set.

2. Task-learner component: A learner (or model) is trained to generalise, by doing task-specific learning over multiple tasks using a small training data set.

2.2.3 Bayesian Probabilistic Perspective of Meta-Learning and MAML

Meta-learning can be considered, from a Bayesian probabilistic perspective, as learning priors over a meta-data set which when combined with a likelihood of a novel data set, given meta-parameters, it improves the efficiency of the learning of task-specific parameters during fine-tuning. This perspective is useful in gaining a greater understanding in the topic, explored in-depth in [31], the contents of which the follow is based upon:
When considering a supervised learning task as a maximum likelihood problem where we wish to maximise the probability of observing model parameters ϕ given a set of data $\mathcal{D} = \{(x_1, y_1), ..., (x_k, y_k)\}$:

$$\begin{aligned}\arg\max_{\phi} \log p(\phi|\mathcal{D}) &= \arg\max_{\phi} \log p(\mathcal{D}|\phi)p(\phi) \\ &= \arg\max_{\phi} \log p(\mathcal{D}|\phi) + \log p(\phi) \\ &= \arg\max_{\phi} \sum_i \log p(y_i|x_i, \phi) + \log p(\phi).\end{aligned}$$

As explained in subsection 1.1.2, with small samples of data the above can be seen to overfit and hence displays the few-shot learning problem.
In meta-learning, we wish to make use of additional information in the form of a meta data-set $\mathcal{D}_{meta} = \{\mathcal{D}_1, ..., \mathcal{D}_n\}$ to help improve the estimation of the task-specific parameters ϕ during adaptation in meta-testing. In doing so, during meta-training, we will learn meta-parameters θ which will serve as a form of information learned across many similar tasks from the meta data set,

which will enable more efficient learning of a similar task with few numbers of data points. Reformatting the above Bayesian perspective of supervised learning into a meta-learning problem, we seek the parameters ϕ^* such that the log-likelihood of observing ϕ given data $\mathcal{D}$ *and* $\mathcal{D}_{meta}$ is maximised:

$$\phi^* := \arg\max_{\phi} \log p(\phi|\mathcal{D}, \mathcal{D}_{meta}).$$

To show that ϕ^* is (approximately) implicitly equivalent to maximising the log-likelihood of observing ϕ given data $\mathcal{D}$ and optimal meta-parameters learned $\theta^* = \arg\max_{\theta} \log p(\theta|\mathcal{D}_{meta})$, we consider:

$$\begin{aligned}
\log p(\phi|\mathcal{D}, \mathcal{D}_{meta}) &= \log \int_{\Theta} p(\phi, \theta|\mathcal{D}, \mathcal{D}_{meta})\, d\theta \\
&= \log \int_{\Theta} p(\phi|\theta, \mathcal{D}, \mathcal{D}_{meta}) p(\theta|\mathcal{D}, \mathcal{D}_{meta})\, d\theta \\
&= \log \int_{\Theta} p(\phi|\theta, \mathcal{D}) p(\theta|\mathcal{D}_{meta})\, d\theta \qquad (\text{assuming } \phi \perp\!\!\!\perp \mathcal{D}_{meta}|\theta\,) \\
&\approx \underbrace{\log p(\phi|\theta^*, \mathcal{D}) + \log p(\theta^*|\mathcal{D}_{meta})}_{\text{adaptation}}
\end{aligned}$$

hence $\phi^* := \arg\max_{\phi} \log p(\phi|\mathcal{D}, \mathcal{D}_{meta}) \approx \arg\max_{\phi} \log p(\phi|\theta^*, \mathcal{D})$.

Chapter 3

Methods

3.1 MAML

3.1.1 Introduction

Model-Agnostic Meta-Learning (MAML) is a meta learning few-shot technique, detailed by Finn et al. in their article *Model-Agnostic Meta-Learning for Fast Adaptation of Deep Learnng*, which can be applied to any machine learning problems optimised using gradient descent. A model is trained on a variety of different tasks so that when introduced to a new, novel task it can perform well when trained only on a few samples. MAML can be applied to such machine learning problems as classification, regression and even reinforcement learning. The intuition behind MAML is that by optimising the parameters of a model such that they are more sensitive to local task-specific changes, when introduced to a N-way few-shot learning problem the model can achieve great generalisation in only a few optimisation steps of gradient descent. Finn et al. go on to describe that, from a feature learning standpoint, optimising a models parameters in this way is equivalent to building an internal representation which is suitable for many tasks. Some of these internal representations are more transferable to new tasks than others, and

it is these internal representations that MAML tries to encourage the emergence of; through learning a model that can quickly adapt to new tasks with sensitive parameters, such that small changes in the model's features result in large decreases of the novel task's loss.

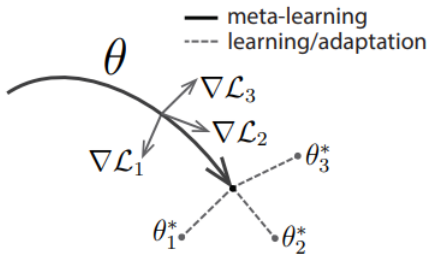


Figure 3.1: MAML algorithm searching for a representation θ through meta-training which quickly adapts to 3 new tasks through gradient based optimisation techniques. *Source: Finn et al.* [6].

3.1.2 Algorithm

The algorithm works as follows. We sample a task (or a batch of tasks) $\mathcal{T}_i \sim p(\mathcal{T})$ and further sample meta-training and meta-testing data sets D_i^{tr} and D_i^{test}, respectively, from these tasks. The model f_θ, parameterised by meta parameters θ, learns optimal task-specific parameters ϕ_i for these tasks using a gradient based optimisation technique, such as stochastic gradient descent, with primary learning rate α and computing loss $\mathcal{L}$ using meta-training data D_i^{tr}. This is known as the inner-loop of the algorithm, where task-specific knowledge is learned, and is the task-learner component as described in Section 2.2.2:

$$\phi_i = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta, D_i^{tr}). \tag{3.1}$$

The inner-loop process can be performed N times for a given task, leading to N-inner gradient updates. We denote the task-specific parameters for task

$\mathcal{T}_i$ after N gradient updates by ϕ_i^N:

$$\phi_i^N = \phi_i^{N-1} - \alpha \nabla_{\phi_i^{N-1}} \mathcal{L}_{\mathcal{T}_i}(f_{\phi_i^{N-1}}, D_i^{tr}) \tag{3.2}$$

For simplicity and ease of understanding, we take $N = 1$ from henceforth. The meta parameters θ are then trained to optimise the performance of f_{ϕ_i} with respects to θ, across the tasks $\mathcal{T}_i$ such that the optimal θ results in the smallest sum of each individual tasks losses using testing data D_i^{test}, at each tasks optimal ϕ_i^N, computed in Equation 3.3. This gives rise to the meta-objective:

$$\arg\min_{\theta} \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\phi_i}, D_i^{test}) = \arg\min_{\theta} \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)}, D_i^{test}).$$

The meta parameters are optimised using stochastic gradient descent[1], in what is known as the outer-loop update, with secondary learning rate β, as follows:

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\phi_i}, D_i^{test}). \tag{3.3}$$

The outer-loop is where across-task knowledge is learned, and is the meta-learner component as described in Section 2.2.2. This whole process is then iteratively repeated. The process of updating these meta-parameters, involving taking a gradient of a gradient and calculating Hessian-vector products, can be efficiently computed using autograd libraries such as TensorFlow or PyTorch. Further details on the this are given in Appendix C.1. The pseudocode of this algorithm can be seen in Figure 3.2. Note that the original notation used θ_i' to represent task-specific parameters, where we used ϕ_i to better distinguish the meta-parameters. In addition to this, we have made

[1]The MAML authors instead use Adam to optimise the meta-parameters, but for ease of understanding we use stochastic gradient descent in the above equation.

clearer distinctions on where the meta-training and testing data is used in the algorithm.

Algorithm 1 Model-Agnostic Meta-Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ with respect to K examples
6: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
7: **end for**
8: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$
9: **end while**

Figure 3.2: Pseudocode for MAML algorithm, as seen in the original article. *Source: Finn et al.* [6].

3.1.3 Empirical Results of Regression and Classification Applications

Finn et al. demonstrated the effectiveness of MAML in both regression and classification tasks, with empirical results using a toy-example in a regression setting and using few-shot image classification benchmark data sets MiniImageNet [32] and Omniglot [33].

Starting with regression, each task $\mathcal{T}_i$ involved regressing a sine wave with varying amplitude and phase between $[1.0, 5.0]$ and $[0, \pi]$ respectively. The model f_θ used was an MLP with 2 hidden layers each with 40 neurons with ReLU activations. $K = 10$ data points were sampled from each task, which were used with mean squared error loss $\mathcal{L}$ and the optimiser Adam to compute each task specific optimal parameters ϕ_i. Adam was also used as the meta-optimiser.

The performance of the MAML trained model was evaluated on varying K

data points of a new task $\mathcal{T}_i$ and then compared to both an oracle and a pretrained model. The meta-learned MAML was shown to quickly adapt to $K = 5$ data points and form a very good approximation with 10 data points, as seen in plots 1 and 2 in Figure 3.3. In addition to this, MAML was able to capture the periodic oscillating nature of the sine wave; with given only 5 data points all coming from the domain $[0, 5]$ the model was still able to produce a good approximation for inputs in domain $[-5, 0]$. In comparison, the pretrained model is seen to have woefully overfit the training data, and cannot produce an equivalently performing model to MAML.

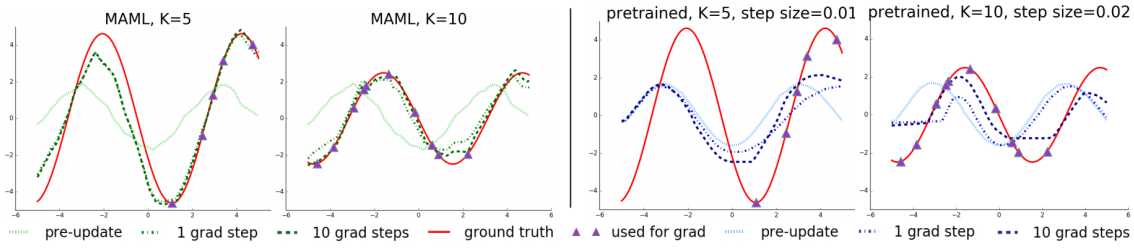


Figure 3.3: In order: plots 1-4. Comparison of MAML trained model against a pretrained model and an oracle. *Source: Finn et al.* [6].

In the classification setting, N-way K-shot classification was performed on benchmark data sets MiniImageNet and Omniglot. A CNN model, consisting of 4 internal modules, was used to classify these images. More specifically, the model included 4 convolutional layers with 64 3×3 filters all followed by batch normalisation [34], a ReLU activation, a pooling layer with 2×2 max-pooling and finally the last layer was fed into a soft-max layer[2] for classification. Cross-entropy was used as the loss function for the models. Data from the Omniglot data set was partitioned into 1200 randomly selected characters for training and the remaining for testing. Data augmen-

[2]A soft-max layer is a layer which assigns multiple classes with a probability of the input being this class.

tation techniques were used, through rotations by multiples of 90 degrees. For the purposes of comparison to previous studies, a standard multilayered perceptron model with hidden layers of sizes 256, 128, 64 and 64, also taking advantage of batch normalisation and ReLU activations, was also trained and used to classify these images.

At the time that the MAML article was published, MAML was shown to produce state-of-the-art results against the aforementioned benchmark classification data sets, which can be seen in Table 3.1.

	5-way Accuracy		20-way Accuracy	
Omniglot (Lake et al., 2011)	1-shot	5-shot	1-shot	5-shot
MANN, no conv (Santoro et al., 2016)	82.8%	94.9%	–	–
MAML, no conv (ours)	**89.7 ± 1.1%**	**97.5 ± 0.6%**	–	–
Siamese nets (Koch, 2015)	97.3%	98.4%	88.2%	97.0%
matching nets (Vinyals et al., 2016)	98.1%	98.9%	93.8%	98.5%
neural statistician (Edwards & Storkey, 2017)	98.1%	99.5%	93.2%	98.1%
memory mod. (Kaiser et al., 2017)	98.4%	99.6%	95.0%	98.6%
MAML (ours)	**98.7 ± 0.4%**	**99.9 ± 0.1%**	**95.8 ± 0.3%**	**98.9 ± 0.2%**

	5-way Accuracy	
MiniImagenet (Ravi & Larochelle, 2017)	1-shot	5-shot
fine-tuning baseline	28.86 ± 0.54%	49.79 ± 0.79%
nearest neighbor baseline	41.08 ± 0.70%	51.04 ± 0.65%
matching nets (Vinyals et al., 2016)	43.56 ± 0.84%	55.31 ± 0.73%
meta-learner LSTM (Ravi & Larochelle, 2017)	43.44 ± 0.77%	60.60 ± 0.71%
MAML, first order approx. (ours)	**48.07 ± 1.75%**	**63.15 ± 0.91%**
MAML (ours)	**48.70 ± 1.84%**	**63.11 ± 0.92%**

Table 3.1: Classification results of MAML used in different N-way K-shot classification settings in comparison to other few-shot learning techniques. *Source: Finn et al.* [6].

3.1.4 FOMAML

Due to the computationally expensive nature of optimising the meta-objective in Equation 3.3 involving calculating Hessian-vector products, Finn et al. proposed a first-order alternative to MAML, FOMAML (First Order Model-Agnostic Meta-Learning), which produced a similar performance to MAML against few-shot benchmark data sets, as seen in Figure 3.1. The first order variation simply omits the second-order derivatives from the meta-

optimisation objective, for which training times are seen to improve by 33%. Additionally, when including additional gradient-steps during the inner-loop optimisation displayed in Equation 3.1, the computational benefit of not having to compute Hessian-vector products becomes apparent. Appendix C.2 details the mathematics behind the outer-loop updates when 2 inner-loop updates are involved.

3.2 MAML++

3.2.1 Introduction

MAML++ [9] is a variant of MAML, proposed by Antoniou et al. in their article *How to Train Your MAML*, which seeks to improve a variety of issues which MAML exhibits. The primary problems of MAML consists of unstable and computationally expensive training. The authors of MAML++ empirically show that their modifications to MAML help remedy these problems, leading to a greater performance against benchmark few-shot data sets.

3.2.2 Weaknesses of MAML

Antoniou et al. go on to explain that the weaknesses of MAML cause problems such as:

1. **Instability during training**: The stability during training is very dependent upon the architecture of the neural network and hyperparameter setup. The exploding/vanishing gradient problem is prominent here, due to the process of optimising the outer-loop involving back-propagating derivatives through the network several times, depending upon the number of inner-loop updates, with no skip-connections[3].

[3]Skip-connections are increasingly-popular modules of CNNs which, as the name suggests, allow for outputs of one layer to entirely skip the following layer and instead become inputs for the succeeding layers.

2. **Restricted generalisation performance**: The authors explain how using running statistics instead of batch statistics when implementing batch-normalisation[4], as used by the authors of MAML, improves converges speed, generalisation performance and stability. This is because using running statistics results in a smoother optimisation "landscape" for learning hyperparameters betas and gammas for batch normalisation.
3. **Computational efficiency problems**: Second order derivatives are required in training MAML. This is extremely costly, and whilst the authors detailed a first-order alternative in FOMAML, this is at the cost of a reduced generalisation performance.
4. **Requirements of computationally inefficient hyperparameter tuning before sufficient performance**: The authors theorise, using recent studies on annealing learning rates, that having a fixed learning rate for both inner and outer-loop optimisation steps produces a weaker generalisation performance and slower optimisation of the model.

3.2.3 Proposed Improvements

The authors then propose solutions to each above problems of MAML individually, which when combined, lead to a stable, automated & improved MAML.

Multi-Step Loss Optimisation (MSL) was used a solution to the problem of instability during training. MSL, used in MAML++, is a modification to how the meta-parameters are optimised during the outer-loop. Instead of optimising meta-parameters using the sum of losses of all tasks trained during

[4]Batch normalisation [34], in short, intends to improve the efficiency of training through normalising activations of neural networks, in a similar fashion to how normalising inputs of neural networks improves training. Batch normalisation uses learnable hyperparameters betas and gammas to normalise these activations.

the inner loop, once all inner-loop gradient steps have been completed, MSL instead uses a weighted sum of each tasks' loss evaluated at each gradient-step performed in the inner loop. This is best understood considering the change in meta-objective equations, between MAML and MAML++:

$$\underbrace{\arg\min_{\theta} \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\phi_i^N}, D_i^{test})}_{\substack{\text{MAML meta-objective,} \\ \text{with } N \text{ inner-loop gradient updates.}}} \rightarrow \underbrace{\arg\min_{\theta} \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \sum_{n=0}^{N} w_n \mathcal{L}_{\mathcal{T}_i}(f_{\phi_i^n}, D_i^{test})}_{\text{MAML++ meta-objective.}},$$

where $\boldsymbol{w} = (w_1, ..., w_N)$ is an N-dimensional weight vector. The weight vector $\boldsymbol{w}$ is initialised with equal weighting, with each gradient-step having equal importance towards contributing to the loss for that task. As training progresses, the weights change such that weights used for earlier gradient-steps loss have lower importance and weights used for later gradient-steps loss have higher importance. This eases training instability due to better gradient propagation throughout the network, easing symptoms of the vanish/exploding gradient problem.

In order to tackle the computationally expensive nature of MAML, the authors demonstrated that by using lower-order gradients during early training and higher-order gradient during the remainder of training, in a process known as derivative-order annealing (DOA), the generalisation performance of MAML does not suffer and the computational efficiency is improved. Antoniou et al. theorised that using first-order derivatives in earlier stages of training provides a solid "pretraining" method for which parameters are learned that do not exhibit gradient explosion/diminishment problems, such that this method also helps to address the first issue of instability during training.

3.2.4 Empirical Results

The results of evaluating the performance of MAML++ against benchmark data sets Omniglot & miniImageNet display a further improvement of generalisation score against MAML on all accounts, as seen in Table 3.2.

Omniglot 20-way Few-Shot Classification		
	Accuracy	
Approach	**1-shot**	**5-shot**
Siamese Nets	88.2%	97.0%
Matching Nets	93.8%	98.5%
Neural Statistician	93.2%	98.1%
Memory Mod.	95.0%	98.6%
Meta-SGD	95.93±0.38%	98.97±0.19%
Meta-Networks	97.00%	–
MAML (original)	95.8±0.3%	98.9±0.2%
MAML (local replication)	91.27±1.07%	98.78%
MAML++	**97.65±0.05%**	**99.33±0.03%**
MAML + MSL	91.53±0.69%	-
MAML + LSLR	95.77±0.38%	-
MAML + BNWB + BNRS	95.35±0.23%	-
MAML + CA	93.03±0.44%	-
MAML + DA	92.3±0.55%	-

Table 3.2: Comparison of MAML++ performance on the Omniglot 20-way few-shot learning data set, with vanilla MAML and other meta-learning algorithms. *Source: Antoniou et al.* [9].

3.3 Rapid Learning or Feature Reuse

3.3.1 Introduction

Raghu et al. [7] investigate whether MAML is learning good meta-learned initialisation parameters (as intended) which when subject to unseen tasks allows for rapid learning, or if MAML is simply learning good feature representations and it is these feature representations which allow for the model to quickly generalise to new tasks (feature reuse). Feature reuse generates good feature representations through learned parameters, which when exposed to unseen tasks can learn optimal task-specific parameters since the meta-initialised parameters are close, as seen in Figure 3.4.

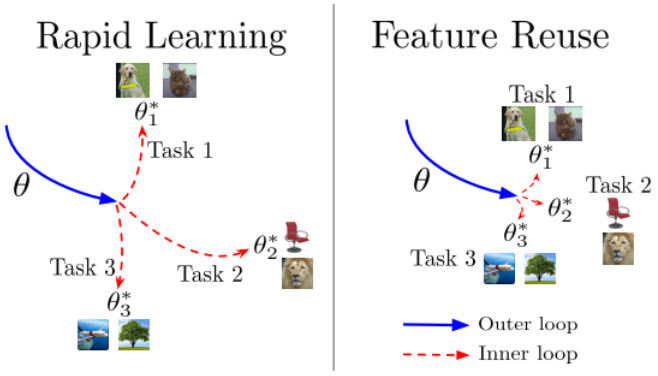


Figure 3.4: Rapid Learning v.s. Feature Reuse: Rapid learning generates parameters which when exposed to unseen tasks in a few-shot setting, can rapidly learning optimal parameters. *Source: Raghu et al. [7].*

3.3.2 Experiments

To these ends, two different analyses are performed using the original 4-layer CNN architecture used in the MAML article for the classification task:

- **Layer freezing experiments**: Parameters of different subsets of layers are fixed after having been trained by MAML, therefore remain unchanged when during test-time when introduced to unseen tasks.
- **Latent representational analyses**: Algorithms Canonical Correlation Analysis (CCA) and Centered Kernel Alignment (CKA) to quantify the similarity of network features and representations are used to investigate the extent to which the inner-loop task-specific adaptation updates have on the network. CCA and CKA output similarity scores between 0 (not similar) and 1 (identical) for layers of neural networks. These are used to compare layers of the CNN before and after the inner loop adaption phase.

3.3.3 Results and Conclusions

Results of the layer freezing experiments are displayed in Table 3.3. It can be seen that even without task-specific inner-loop adaptation very similar gen-

eralisation performances can be achieved. This implies that the meta-learned initialisation parameters already have excellent feature representations, and that test-time task specific training is not necessary for a good performance.

Freeze layers	MiniImageNet-5way-1shot	MiniImageNet-5way-5shot
None	46.9 ± 0.2	63.1 ± 0.4
1	46.5 ± 0.3	63.0 ± 0.6
1,2	46.4 ± 0.4	62.6 ± 0.6
1,2,3	46.3 ± 0.4	61.2 ± 0.5
1,2,3,4	46.3 ± 0.4	61.0 ± 0.6

Table 3.3: Results of layer freezing experiments. *Source: Raghu et al. [7].*

Results of the latent representational analyses, displayed in Figure 3.5, show high similarity in feature representation before and after inner loop adaption phase for all layers of the neural network except the head (final output layer). These results reinforce the conclusions of the layer freezing experiments, that good feature representations are learned in MAML rather than initialisations which allow for rapid learning.

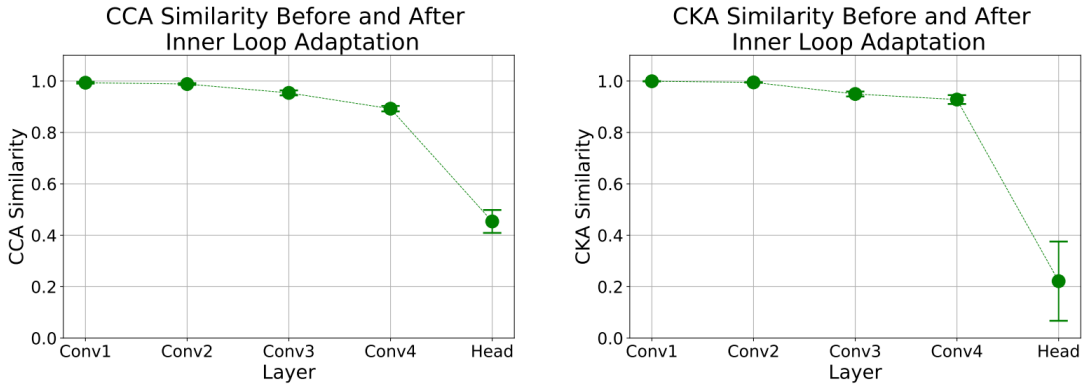


Figure 3.5: Results of CCA and CKA similarity experiments. *Source: Raghu et al. [7].*

3.3.4 ANIL

Following on from the prior experimental results, Aniruddh Raghu et al. go on to propose two modifications to MAML named ANIL (Almost No Inner Loop). ANIL removes the inner loop updates for all layers except the head (final layer) of the network in an attempt to capitalise on the experimental results that the body parameters of the network do not change much during the inner loop updates, resulting in computational and generalisation performance benefits.

In addition to a significant computational improvement during training, ANIL is shown to have a similar performance against Omniglot and MiniImageNet data sets to MAML, and even having a 2% accuracy improvement in 20-way 1 and 5-shot classification tasks as displayed in Table 3.4.

Method	Omniglot-20way-1shot	Omniglot-20way-5shot	MiniImageNet-5way-1shot	MiniImageNet-5way-5shot
MAML	93.7 ± 0.7	96.4 ± 0.1	46.9 ± 0.2	63.1 ± 0.4
ANIL	96.2 ± 0.5	98.0 ± 0.3	46.7 ± 0.4	61.5 ± 0.5

Table 3.4: ANIL Omniglot and MiniImageNet results. *Source: Raghu et al. [7].*

3.3.5 NIL

The authors then decided to investigate the effectiveness of the head of the network, at the fine-tuning stage, after good features have already been learned from MAML and ANIL. To do this, they developed the No Inner Loop (NIL) algorithm:

1. Train few-shot learning model with ANIL/MAML.
2. During the fine-tuning stage, remove the head (final layer) of the trained network. Then for each task:
    - Pass K labelled examples, of each class from the novel data set, through the body of the network to get penultimate layer repre-

sentations.

- Then for an additional novel "testing" data point, compute cosine similarities between its penultimate layer representation and those of the K labelled examples from the set above. Use these similarities to weight the support set labels (seen in Vinyals et al. [32]).

The results of the NIL algorithm, displayed in Table 3.5, show excellent classification accuracy despite the models having their final layer removed and no test-time task specific adaptation. This implies that primarily the body of the network learned during training, and the feature representations that they learn are the essential part of model and that the head of the model is not important during test time.

Method	Omniglot-20way-1shot	Omniglot-20way-5shot	MiniImageNet-5way-1shot	MiniImageNet-5way-5shot
MAML	93.7 ± 0.7	96.4 ± 0.1	46.9 ± 0.2	63.1 ± 0.4
ANIL	96.2 ± 0.5	98.0 ± 0.3	46.7 ± 0.4	61.5 ± 0.5
NIL	96.7 ± 0.3	98.0 ± 0.04	48.0 ± 0.7	62.2 ± 0.5

Table 3.5: Comparison of NIL (with ANIL training) to ANIL and MAML in few-shot classification of Omniglot and MiniImageNet data sets. *Source: Raghu et al. [7].*

Chapter 4

Experimental Studies

4.1 Modified Toy Sinusoid Experiment

4.1.1 Introduction

The first experiment conducted was investigating the effectiveness of MAML when faced with a modified version of the toy sinusoid problem, originally demonstrated in [6] and described in section 3.1.3. I hypothesised that the distribution from which the sinusoidal tasks were sampled from resulted in tasks which were too similar, and wanted to experiment with using an additional parameter for which the tasks can better differentiate between themselves with. This additional parameter came of the form of multiplying these sinusoidal curves, with varying amplitude and phase, by x^{pow}. This additional parameter *pow* is seen to extend the original distributions of vanilla sine curves to a distribution containing (in additional to the original family of sinusoidal curves[1]) a family of sine curves with varying amplitudes, phases *and* a potential additional turning point through x^{pow}, where $power \in \{0, 1\}$.

[1]Note how that, when $power = 0$, the task reduces to a simple sine curve with varying amplitude and phase. When $pow = 1$, an additional turning point is introduced; hence increasing the variability of tasks which can be drawn. This is more easily seen in the next subsection.

To these ends, I implemented the MAML algorithm in Python, using PyTorch, with methods in MAML++ for computational benefits in addition to combating any gradient issues present. The code has been implemented using a Google Colab notebook, making use of the free GPU support, and can be found in the follow link: `https://colab.research.google.com/drive/1xtm-BwkcQMsVBQ9QkaLDeq9afY4FPXbj?usp=sharing`.

4.1.2 Problem Framing and Setup

The original sinusoidal tasks were sampled from a distribution which resulted in functions such as:

$$f(x) = amplitude \times \sin(x + phase),$$

where $amp \in [0.1, 5.0]$ and $phase \in [0, \pi]$.

My modified experiment samples sinusoidal tasks from a distribution which resulted in functions such as:

$$f(x) = amplitude \times x^{power} \times \sin(x + phase),$$

where $amp \in [0.1, 5.0]$, $phase \in [0, \pi]$ and $power \in \{0, 1\}$. This extra *power* parameter allows for the creation of tasks which, in comparison to the original sinusoidal experiments, allows for greater variability in the distribution of curves which can be sampled.

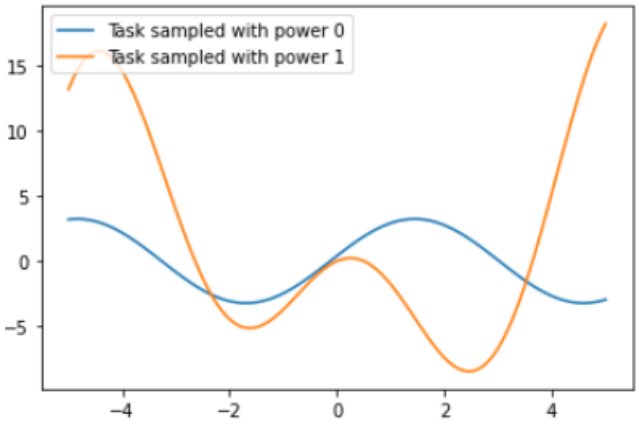


Figure 4.1: Curves for two sampled tasks with differing *power* values, with amplitudes and phases 3.23, 0.13 and 3.73, 2.64, respectively.

In order to achieve greater computational efficiency and to tackle and exploding/vanishing gradient problems present, I implemented DOA, as described in Section 3.2.3, such that earlier stages of meta-training only included first-order derivatives (the first 2,500 iterations of training) and latter stages were allowed to include second-order derivatives (the following 2,500 iterations of training). Learning rates $\alpha = 0.02$ and $\beta = 0.02$ were used, conceived through a trail-and-error process of experimenting with a variety of varying magnitudes of these learning rates on the base data set. Adam was used in optimising the meta-objective, whilst stochastic gradient descent was used in the inner-loop task-specific optimisation. The neural network model trained had identical structure to that of which was used in the original toy sinusoidal experiments; 2 hidden layers of size 40, with ReLU nonlinearities. The losses of 1,000 tasks, computed using mean squared error, were used for each outer-loop update, $\mathcal{T}_i$, of which each had a sample of 10 data points for meta-training, D^{tr}, and an additional 10 data points for meta-testing, D^{test}. All data used for training was sampled in the domain $x \in [-5, 5]$.

4.1.3 Empirical Results

Following training, I sampled a number of tasks from the new distribution with equal number of tasks holding the two *power* parameter values. I then fine-tuned each task, involving optimising the learned MAML meta-parameters θ with 10 gradient updates using Adam, using the same $K = 10$ sampled training data points for each epoch of training. For comparison, I pre-trained a model, which entailed training 500 neural networks of identical structure to that used in the MAML model on 500 individual tasks each.

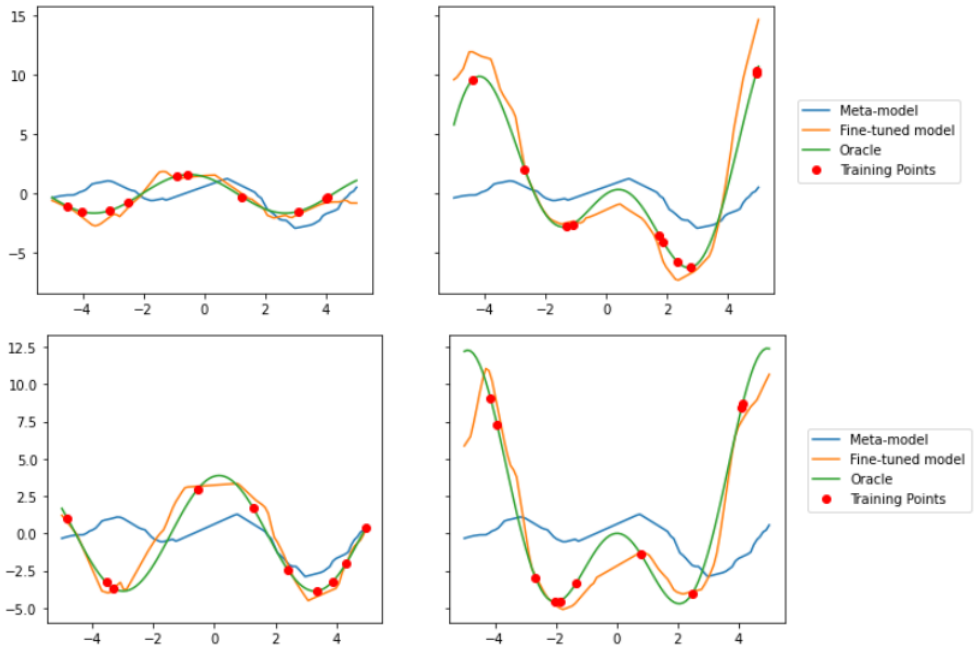


Figure 4.2: A comparison of a random selection of tasks with varying parameters displaying their performance once fine-tuned. The first column displays tasks with *power* parameter 0, and the second column tasks with *power* parameter 1.

Figure 4.2 displays a random selection of four sampled tasks with differing parameters, displaying their performance once fine-tuned, as described above. Plots in the first column have sampled tasks with *power* parameter 0, and the second column have tasks with *power* parameter 1. The red dots

on the plots represent the training data used in fine-tuning. The blue meta-model curve represents the learned MAML parameters. The green oracle curve represents the true underlying sampled sinusoidal curve distribution. The yellow fine-tuned model curve displays the model after fine-tuning the meta-model with the training data. From inspecting the plots, the MAML parameters seem to form a good foundation for each task, as the model was able to quickly generalise to the underlying sampled tasks distribution. This can be seen in all plots, as the fine-tuned curves are relatively close to the oracle curves. I was impressed to see that the model was easily able to identify the most probable underlying *power* parameter to the task, and hence be able to form the general shape of the curve.

Unfortunately, the performance of the MAML parameters in generalising well to a specific task with small amounts of data seems to be heavily reliant on the location of data sampled. This can be seen in Figure 4.3, as the fine-tuned model for the same task generalises better with data evenly distributed across the sampling domain, instead of being concentrated to a single spot. In addition to this, the generalisation performance does indeed seem to be weaker than that obtained in the original toy sinusoidal experiments by Finn et al., which can be seen in Figure 3.3, as the fine-tuned model (green dotted curve) is a closer representation of the underlying sampled task distribution (red solid curve) than that obtained in my experimental results in Figure 4.2. I believe this is an expected result, though, as the greater variance of sampled tasks in my modified experiment is a more complex, difficult problem and perhaps requires a more complex neural network model architecture, such as a deeper network, to capture such complex relationships between input and output, in addition to better training conditions through finding more optimal hyperparameters and a greater number of training iterations.

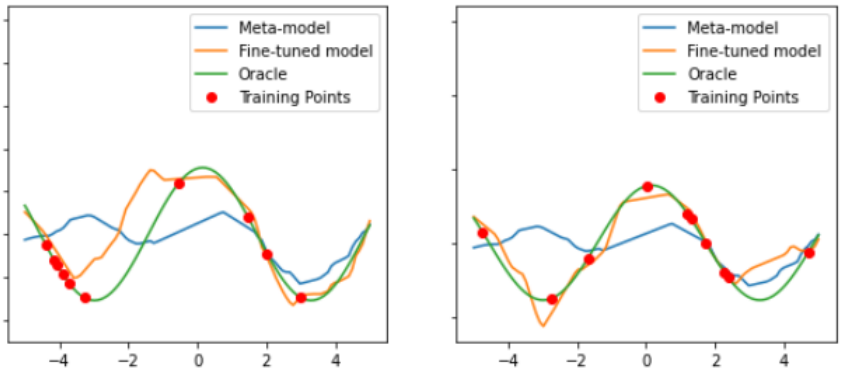


Figure 4.3: Comparison of how the distribution of training data used in fine-tuning affects the generalisation of the resulting few-shot learned model, for the same task. Plot 1 displays more concentrated data in a single location, whereas plot 2 displays more evenly distributed data.

To form a comparison from which we can evaluate the performance of the MAML-trained parameters against, I repeated the process by which Finn et al. use a multi-task method to generate a model parametrised by the average parameters of many trained models on individual tasks with a surplus amount of data. This involved training 500 models, each on their own task, with identical neural network architecture to that previously mentioned, from scratch. The parameters of these 500 models are then averaged to form the pre-trained model for comparison. This pre-trained model is then fine-tuned, using the exact same process used in MAML fine-tuning, with the same data points used to fine-tune the same tasks with MAML. As shown in Figure 4.4, the pre-trained model does not compare in performance to the MAML-trained model, failing to adapt to the task at hand with the small amount of data available for fine-tuning.

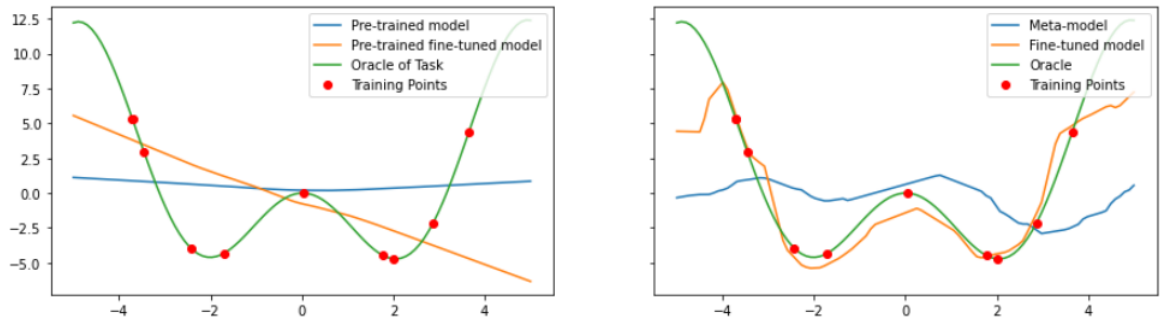


Figure 4.4: Comparison of the pre-trained model against MAML-trained model.

4.1.4 Conclusion

MAML is an effective yet moderately simple tool for generating parameters with which tasks can rapidly generalise with few amounts of data. Despite adding an extra parameter for which the tasks can better differentiate between themselves, through the *power* parameter, the performance of which these tasks generalise seems to be marginally affected, in comparison to the original sinusoidal experiment. Though experimental results seem to suggest that the performance of the learned MAML meta-parameters are dependent on the location of the training data used during fine-tuning, I believe that this result is largely expected to happen. Due to time constraints and other prioritisation in this project, I was not able to selectively pick better hyper-parameters for meta-training the model (such as the inner and outer-loop learning rates) or be able to do meta-training for a larger number of iterations than 5,000. Doing so would have resulted in potentially an extra level of improved performance.

4.2 MAML Applied to Fewshot-CIFAR100

The secondary experimental study conducted involved implementing MAML and applying it to the Fewshot-CIFAR100 dataset, with a focus on 5-way 5-

shot learning. This implementation made use of the meta-learning software library *learn2learn* [35]. Learn2learn is built on top of PyTorch, and enables fast prototyping and correct reproducibility of meta-learning algorithms, such as, but not limited to, MAML. The Python code written to implement this experimental study can be seen at `https://colab.research.google.com/drive/14u7HHPfYjz8eWNvndFbGL7_INKIQykKM?usp=sharing`.

4.2.1 Fewshot-CIFAR100 Dataset

The Fewshot-CIFAR100 (CIFAR-FS) dataset, designed by Bertinetto et al. [36], is an adapted dataset from the CIFAR-100 dataset [37] specifically for few-shot learning scenarios. CIFAR-FS consists of a random sample from CIFAR-100, with 100 different classes each containing 600 RGB images of size 32 by 32 pixels. These classes are split into 64 base classes, 16 validation classes and 20 novel classes. Such classes in this data set consist of a wide range of different animals, insects, objects and people.

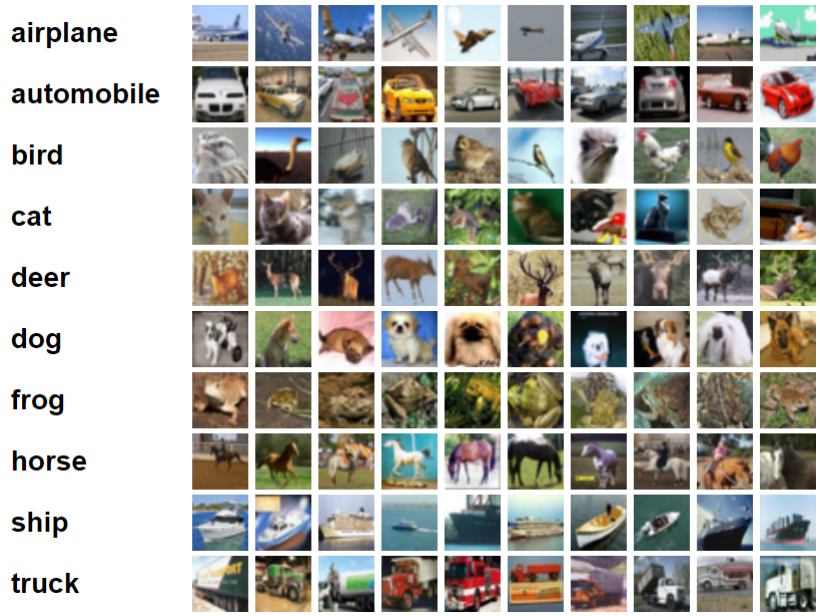


Figure 4.5: Randomly sampled images from 10 classes within the CIFAR-100 dataset. *Source: Alex Krizhevsky [8].*

4.2.2 Empirical Results

In implementing MAML, I used a similar CNN architecture to that which is used in similar few-shot vision benchmarks; a 4-layered CNN with 3×3 convolutions, 32 filters, ReLU non-linearity activations, followed by a fully connected final layer with an output size of 5 (to match the 5-way learning scenario). An inner and outer loop learning rate of 0.5 and 0.003, respectively was used during MAML. Only 1 inner-loop gradient update was used. Adam was used to optimise the meta-parameters during the outer-loop update. After training for 10,000 iterations, the model was able to achieve a 64% accuracy in 5-way 5-shot classification after fine-tuning was completed.

In order to determine the effectiveness of my implementation of MAML from these results, Table 4.1 presents a comparison of empirical results of alternative methods used to classify the CIFAR-FS data set under a 5-way 5-shot learning scenario. It can be seen that my implementation of MAML yields the weakest accuracy results. As research has progressed in few-shot learning over the years, the performance of newly developed models against this few-shot benchmark data set has greatly increased. Bertinetto et al.'s 2017 implementation of MAML achieved an accuracy of $71.5\% \pm 1.0$. Their implementation, however, uses additional regularisation techniques, such as dropout, and a different and considerably more complex neural network architecture to the model used in my implementation[2]. The SKD-GEN1 model, implemented by Rajasegaran et al. [10] in their 2020 article, involves improving the representation capacity of deep neural networks. The topic of this work is under learning good feature embedding, which is shown to outperform classical meta-learning techniques, which entails, in short, transforming

[2]The full details on the differing architecture structure can be found in [36].

features from original feature space to a better space to facilitate effective learning. The accuracy of this model is seen to achieve a groundbreaking accuracy of $88.9\% \pm 0.6$.

5-way 5-shot CIFAR-FS Benchmark Results		
Method	Year	Accuracy (%)
MAML - My implementation	n/a	64
MAML - Bertinetto et al. implementation [36]	2017	71.5 ± 1.0
Boosting [38]	2019	86.0 ± 0.2
RFS-distill [39]	2020	86.9 ± 0.5
SKD-GEN1 [10]	2020	88.9 ± 0.6

Table 4.1: A Sample of 5-way 5-shot CIFAR-FS Benchmark Results. *Source: Results taken from Rajasegaran et al. [10]*

4.2.3 Conclusion

As stated in the reported findings of MAML++, the performance of MAML is highly sensitive to the architecture structure of the neural network model used. This is particularly so when using sophisticated structures such as CNNs, and apparent in the preceding empirical results, in comparing my implementation of MAML with Bertinetto et al.'s implementation accuracy. Since the implementation of MAML was on the same Fewshot-CIFAR100 dataset, I believe the only reason for the difference in accuracy between the two implementations is from the differing neural network architecture structure and hyperparameters. With additional time, as with the first empirical study, a greater performance could have been achieved from searching for better hyperparameters, including structure of the CNN itself. When comparing the empirical results of my implementation of MAML with more recent implementations of more advanced models, it is clear that MAML is not the best technique for few-shot learning anymore. The main advantages of MAML lies within its relatively (when comparing and contrasting to more

modern techniques) simple method, which is more easily understandable to non-experts of the few-shot learning field.

Chapter 5

Conclusions

5.1 Conclusions

This project has achieved its primary aim of investigating how MAML, through optimisation-based meta-learning, can help aid few-shot learning problems. This was done through conducting a thorough study into preliminary material on deep learning and meta learning, discussing related academic literature on MAML and finally through conducting two experimental studies implementing the MAML algorithm itself. MAML is shown to be a relatively simple solution to the few-shot learning problem, with apparent moderate effectiveness, even when faced with additional difficulties than originally shown, as shown in the primary experimental study. The secondary experimental study demonstrates the fundamental weakness of MAML, in the sensitivity of the neural network architecture, and highlights how MAML is perhaps a technique of the past, with new methods eclipsing MAML's performance against few-shot benchmark data sets.

5.2 Limitations

MAML is certainly not perfect; it comes with its own distinct weaknesses which were made apparent through the abundance of suggested improvements in MAML++. In addition to this, the literature relating to the perspective of MAML being rapid learning or good feature reuse raises significant open-ended questions on how MAML should be viewed. The weakness of sensitive neural network architecture is shown in the secondary experimental study, as without sufficient time to adequately perform training of the meta-parameters through MAML with several variants of different hyperparameters, including variants of neural network architecture structure, the performance of the meta-parameters learned can be significantly weakened. In addition to this, more modern techniques are shown to drastically out-perform MAML on few-shot benchmark data sets.

5.3 Future Work

Although a concentrated focus was put on MAML during this project, the reality is that MAML is just one solution to the few-shot learning problem. Future work involves looking at alternative methods of solution to the few-shot learning problems, and meta-learning is just one of them. Wang et. al. [5] demonstrate in their survey on few-shot learning the sheer abundance of alternative methods and solutions that exist that can be used in few-shot learning scenarios. More recent studies on learning feature embedding learning show very promising results, as demonstrated in Table 4.1 where the SKD-GEN1 model, which uses feature embedding, has cutting edge empirical results and provides a promising direction for future work. Tian et al. [39] provided the inspiration for this model, from their article *Rethinking Few-Shot Image Classification: a Good Embedding Is All You Need?*, who state

that they believe their findings "*motivate a rethinking of few-shot image classification benchmarks and the associated role of meta-learning algorithms*".

Appendix A

Multilayer Perceptrons

A.1 Non-linear and Linear Activation Functions

In demonstrating how through not using non-linear activation functions in neural networks results in simple affine transformations from inputs to outputs, we consider a 2-layer multilayer perceptron. For the first hidden layer, we have

$$\boldsymbol{H}_1(\boldsymbol{x}) = \boldsymbol{W}_1^T \boldsymbol{x} + \boldsymbol{b}_1.$$

The output layer is given as

$$\begin{aligned}
\boldsymbol{H}_2(\boldsymbol{H}_1(\boldsymbol{x})) &= \boldsymbol{W}_2^T \boldsymbol{H}_1(\boldsymbol{x}) + \boldsymbol{b}_2 \\
&= \boldsymbol{W}_2^T (\boldsymbol{W}_1^T \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 \\
&= \boldsymbol{W}_2^T \boldsymbol{W}_1^T \boldsymbol{x} + \boldsymbol{W}_2^T \boldsymbol{b}_1 + \boldsymbol{b}_2 \\
&= \boldsymbol{W}_*^T \boldsymbol{x} + \boldsymbol{b}_*
\end{aligned}$$

where $\boldsymbol{W}_*^T = \boldsymbol{W}_2^T \boldsymbol{W}_1^T$ and $\boldsymbol{b}_* = \boldsymbol{W}_2^T \boldsymbol{b}_1 + \boldsymbol{b}_2$. Hence, without non-linear activation functions, the output of a 2-layer multilayered perceptron is a simple affine transformation of the input. This theory can be trivially extended to any arbitrarily layered multilayered perceptron.

Appendix B

Autodiff

B.1 Reverse and Forward Mode Autodifferentiation Example Calculations

For further clarification on how autodiff works, a simple example is given for both forward and reverse mode using the function $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ in table B.1. In this example, we find the partial derivative of f with respects to input x_1. The same can be done in a similar fashion for x_2. Note how in forward mode, gradients of variables with respects to inputs, $\dot{v}$, are propagated forwards through the graph, from input variables to output variables. In reverse mode, gradients of outputs with respects to variables, $\bar{v}$, are propagated backwards through the graph, from outputs to inputs.

	Forward Trace	Derivative Trace	
	Both Modes	Forward Mode	Backward Mode
Input variables	$v_{-1} = x_1$	$\dot{v}_{-1} = \frac{\partial v_{-1}}{\partial x_1} = 1$	$\bar{v}_{-1} = \frac{\partial v_1}{\partial v_{-1}}\bar{v}_1 + \frac{\partial v_2}{\partial v_{-1}}\bar{v}_2 = \frac{1}{v_1}\bar{v}_1 + v_0\bar{v}_2$
	$v_0 = x_2$	$\dot{v}_0 = \frac{\partial v_0}{\partial x_1} = 0$	$\bar{v}_0 = \frac{\partial v_2}{v_0}\bar{v}_2 + \frac{\partial v_3}{v_0}\bar{v}_3 = v_{-1}\bar{v}_2 + \cos(v_0)\bar{v}_3$
Intermediary variables	$v_1 = \ln(v_{-1})$	$\dot{v}_1 = \frac{\partial v_1}{\partial v_{-1}}\dot{v}_{-1} = \frac{1}{v_{-1}}$	$\bar{v}_1 = \frac{\partial v_4}{\partial v_1}\bar{v}_4 = \bar{v}_4$
	$v_2 = v_{-1} \times v_0$	$\dot{v}_2 = \frac{\partial v_2}{\partial v_0}\dot{v}_{-1} + \frac{\partial v_2}{\partial v_0}\dot{v}_0 = 0$	$\bar{v}_2 = \frac{\partial v_4}{\partial v_2}\bar{v}_4 = \bar{v}_4$
	$v_3 = \sin(v_0)$	$\dot{v}_3 = \frac{\partial v_3}{\partial v_0}\dot{v}_0 = \cos(v_0) \times 0 = 0$	$\bar{v}_3 = \frac{\partial v_5}{\partial v_3}\bar{v}_5 = -\bar{v}_5$
	$v_4 = v_1 + v + 2$	$\dot{v}_4 = \frac{\partial v_4}{\partial v_1}\dot{v}_1 + \frac{\partial v_4}{\partial v_2}\dot{v}_2 = \dot{v}_1$	$\bar{v}_4 = \frac{\partial v_5}{\partial v_4}\bar{v}_5 = \bar{v}_5$
	$v_5 = v_4 - v_3$	$\dot{v}_5 = \frac{\partial v_5}{\partial v_3}\dot{v}_3 + \frac{\partial v_5}{\partial v_4}\dot{v}_4 = \dot{v}_1$	$\bar{v}_5 = \frac{\partial f}{\partial v_5}\bar{f} = 1$
Output variables	$f(x_1, x_2) = v_5$	$\dot{f} = \frac{\partial f}{\partial v_5}\dot{v}_5$	$\bar{f} = \frac{\partial f}{\partial f} = 1$

Table B.1: Reverse and forward mode auto-differentiation calculations for $f(x_1, x_2) = \ln(x_1) + x_1x_2 - \sin(x_2)$. Such tables are known as an evaluation traces of elementary operations, or Wengert lists.

B.2 Jacobin-Vector Products in Consideration of Reverse and Forward Mode Complexity

Reverse-mode autodiff is computationally more efficient than forward-mode autodiff when dealing with a large number of inputs and smaller numbers of outputs. This can be seen by considering a problem of m input variables and n output variables. By computing Jacobian-vector products for forward-mode automatic differentiation through initialising seeds for all inputs, $\dot{x}_1 =$

$t_1, ..., \dot{x}_m = t_m$, we can see that after one forward pass we can compute $\dot{f}_i$ for all $i \in \{1, ..., n\}$ by

$$\dot{f}_i = t_1 \frac{\partial f_i}{\partial x_1} + ... + t_m \frac{\partial f_i}{\partial x_m}.$$

In matrix form, we have

$$\begin{pmatrix} \dot{f}_1 \\ \vdots \\ \dot{f}_n \end{pmatrix} = \begin{pmatrix} t_1 \frac{\partial f_1}{\partial x_1} + ... + t_m \frac{\partial f_1}{\partial x_m} \\ \vdots \\ t_1 \frac{\partial f_n}{\partial x_1} + ... + t_m \frac{\partial f_n}{\partial x_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & ... & \frac{\partial f_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & ... & \frac{\partial f_n}{\partial x_m} \end{pmatrix} \begin{pmatrix} t_1 \\ \vdots \\ t_m \end{pmatrix} = \boldsymbol{J} \cdot \boldsymbol{t} \tag{B.1}$$

where $\boldsymbol{J}$ is the Jacobian matrix, $\boldsymbol{t}$ the vector of initialised seeds and $\boldsymbol{J} \cdot \boldsymbol{t}$ the Jacobian-vector product. Similarly, for reverse-mode, can be done by initialising seeds $\bar{f}_1 = s_1, ..., \bar{f}_n = s_n$ and we can see that after one backwards pass we can compute $\bar{x}_i$ for all $i \in \{1, ..., m\}$ by

$$\bar{x}_i = s_1 \frac{\partial f_1}{\partial x_i} + ... + s_n \frac{\partial f_n}{\partial x_i}.$$

In matrix form, we have

$$\begin{pmatrix} \bar{x}_1 \\ \vdots \\ \bar{x}_m \end{pmatrix} = \begin{pmatrix} s_1\frac{\partial f_1}{\partial x_1} + ... + s_n\frac{\partial f_n}{\partial x_1} \\ \vdots \\ s_1\frac{\partial f_1}{\partial x_m} + ... + s_n\frac{\partial f_n}{\partial x_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & ... & \frac{\partial f_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & ... & \frac{\partial f_n}{\partial x_m} \end{pmatrix}^T \begin{pmatrix} s_1 \\ \vdots \\ s_n \end{pmatrix} = \boldsymbol{J}^T \cdot \boldsymbol{s}$$

where $\boldsymbol{J}^T$ is the transposed Jacobin matrix, $\boldsymbol{s}$ the vector of initialised seeds and $\boldsymbol{J}^T \cdot \boldsymbol{s}$ is the Jacobin-vector product. When $m \gg n$, the Jacobin is an n by m matrix with many more columns than rows. In reverse-mode autodiff, you would need to multiply the transposed Jacobin matrix using n seeds (involving n backwards passes), in comparison to forward-mode where you would need to multiply the Jacobin matrix using m seeds (involving m forward passes). Since $m \gg n$, reverse-mode autodiff is much more efficient[1] and hence preferred in these types of problems.

[1]If you are familiar with Big O notation, the complexity of forward and reverse mode autodiff is $\mathcal{O}(\dim(\boldsymbol{x})) * \mathcal{O}(\boldsymbol{f})$ and $\mathcal{O}(\dim(\boldsymbol{f})) * \mathcal{O}(\boldsymbol{f})$, respectively.

Appendix C

MAML Optimisation

C.1 Hessian-Vector Products

In performing the meta-optimisation required in MAML, Hessian-vector products are involved. This can be seen as follows: Let $U(\theta, D^{tr}) := \phi = \theta - \alpha \nabla_\theta \mathcal{L}(\theta, D^{tr})$ denote the update rule used for optimising ϕ.
The meta-optimisation objective is given as

$$\min_\theta \mathcal{L}(\phi, D^{test}) = \min_\theta \mathcal{L}(U(\theta, D^{tr}), D^{test}).$$

Due to the gradient-based optimisation techniques we are depending upon, we require $\frac{d}{d\theta}\mathcal{L}(\phi, D^{test})$ in order to update θ. This can be broken down into a Hessian-vector product as follows:

$$\begin{aligned}\frac{d}{d\theta}\mathcal{L}(\phi, D^{test}) &= \frac{d}{d\theta}\mathcal{L}(U(\theta, D^{tr}), D^{test}) \\ &= \underbrace{\nabla_\Theta \mathcal{L}(\Theta, D^{test})|_{\Theta = U(\theta, D^{tr})}}_{(1)} \underbrace{\frac{d}{d\theta} U(\theta, D^{tr})}_{(2)} \quad \text{(via chain rule)}.\end{aligned}$$

(1) is a row vector which can be computed through a single backwards pass of the network, when setting parameters to Θ then differentiating loss $\mathcal{L}$ with respects to Θ. The Hessian matrix (2) is obtained through differentiating the update rule $U(\theta, D^{tr}) = \theta - \alpha \nabla_\theta \mathcal{L}(\theta, D^{tr})$ with respects to θ:

$$\frac{d}{d\theta} U(\theta, D^{tr}) = \mathbb{I} - \alpha \frac{d}{d\theta^2} \mathcal{L}(\theta, D^{tr}),$$

where $\mathbb{I}$ denotes the identity matrix.

C.2 The Case of 2-inner Gradient Steps

First, considering 2-inner gradient steps, we have for the first inner-gradient update, U_1:

$$U_1(\theta, D^{tr}) := \phi^{(1)} = \theta - \alpha \nabla_\theta \mathcal{L}(\theta, D^{tr}).$$

The second gradient update, U_2, can be decomposed into a function of the original meta-parameters θ and the once-updated task-specific parameters $\phi^{(1)}$:

$$\begin{aligned} U_2(\phi^{(1)}, D^{tr}) := \phi^{(2)} &= \phi^{(1)} - \alpha \nabla_{\phi^{(1)}} \mathcal{L}(\phi^{(1)}, D^{tr}) \\ &= \theta - \alpha \nabla_\theta \mathcal{L}(\theta, D^{tr}) - \alpha \nabla_{\phi^{(1)}} \mathcal{L}(\phi^{(1)}, D^{tr}). \end{aligned}$$

To perform the outer-loop update, we require the derivatives of the loss $\mathcal{L}$ computed using predictions from the network parameterised by task-specific adapted parameters $\phi^{(2)}$, with respects to meta-parameters θ:

$$\begin{aligned}\nabla_\theta \mathcal{L}(\phi^{(2)}, D^{test}) &= \nabla_\theta \mathcal{L}(U_2(\phi^{(1)}, D^{tr}), D^{test}) \\ &= \nabla_\theta \mathcal{L}(U_2(U_1(\theta, D^{tr}), D^{tr}), D^{test}) \\ &= \underbrace{\nabla_\Theta \mathcal{L}(\Theta, D^{test})|_{\Theta = U_2(\phi^{(1)}, D^{tr})}}_{(1)} \underbrace{\nabla_\theta U_2(U_1(\theta, D^{tr}), D^{tr})}_{(2)}.\end{aligned}$$

Similarly as in Appendix C.1, (1) is a row vector which can be computed through a single backwards pass of the network, when setting parameters to $\Theta = U_2(\phi^{(1)}, D^{tr})$, then differentiating the loss $\mathcal{L}$ with respects to Θ. We can compute (2) as follows:

$$\begin{aligned}\nabla_\theta U_2 &= \nabla_\theta(\theta - \alpha \nabla_\theta \mathcal{L}(\theta, D^{tr}) - \alpha \nabla_{\phi^{(1)}} \mathcal{L}(\phi^{(1)}, D^{tr}) \\ &= \mathbb{I} - \alpha \nabla^2_\theta \mathcal{L}(\theta, D^{tr}) - \alpha \nabla^2_{\phi^{(1)}} \mathcal{L}(\phi^{(1)}, D^{tr}) \frac{d\phi^{(1)}}{d\theta}\end{aligned}$$

Bibliography

[1] Gonzalo Medina. Diagram of an artificial neural network, 2013. `https://tex.stackexchange.com/a/132471`, Last accessed on 2020-08-29.

[2] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[3] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[4] MathWorks. Convolutional neural network, 2020. `https://uk.mathworks.com/solutions/deep-learning/convolutional-neural-network.html`, Last accessed on 2020-08-09.

[5] Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM Computing Surveys (CSUR)*, 2019.

[6] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *arXiv preprint arXiv:1703.03400*, 2017.

[7] Aniruddh Raghu, Maithra Raghu, Samy Bengio, and Oriol Vinyals. Rapid learning or feature reuse? towards understanding the effectiveness of maml. *arXiv preprint arXiv:1909.09157*, 2019.

[8] Alex Krizhevsky. The cifar-100 dataset, 2020. `https://www.cs.toronto.edu/~kriz/cifar.html`, Last accessed on 2020-08-26.

[9] Antreas Antoniou, Harrison Edwards, and Amos Storkey. How to train your maml. *arXiv preprint arXiv:1810.09502*, 2018.

[10] Jathushan Rajasegaran, Salman Khan, Munawar Hayat, Fahad Shahbaz Khan, and Mubarak Shah. Self-supervised knowledge distillation for few-shot learning. *arXiv preprint arXiv:2006.09785*, 2020.

[11] DeepMind. Alphago the story so far, 2020. `https://deepmind.com/research/case-studies/alphago-the-story-so-far`, Last accessed on 2020-08-03.

[12] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[13] Tesla. Future of driving, 2020. `https://www.tesla.com/en_GB/autopilot`, Last accessed on 2020-08-03.

[14] Wikipedia, the free encyclopedia. Facebook-cambridge analytica data

scandal, 2020. https://en.wikipedia.org/wiki/Facebook%E2%80%93Cambridge_Analytica_data_scandal, Last accessed on 2020-08-03.

[15] The Guardian. Cambridge analytica scandal 'highlights need for ai regulation', 2018. https://www.theguardian.com/technology/2018/apr/16/cambridge-analytica-scandal-highlights-need-for-ai-regulation, Last accessed on 2020-08-03.

[16] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[18] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA, 1969.

[19] Andrew Ng. Cs 229: Machine learning : Supervised learning. University Lecture, 2019. http://cs229.stanford.edu/.

[20] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[21] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[22] Atılım Günes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *The Journal of Machine Learning Research*, 18(1):5595–5637, 2017.

[23] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[24] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[25] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[26] Mike Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 2018.

[27] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia:

A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[28] Fei-Fei Li. Cs 231n: Convolutional neural networks for visual recognition. University Lecture, 2020. `http://cs231n.stanford.edu/`.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[30] Xiaoxu Li, Zhuo Sun, Jing-Hao Xue, and Zhanyu Ma. A concise review of recent few-shot meta-learning methods. *arXiv preprint arXiv:2005.10953*, 2020.

[31] Chelsea Finn. Cs 330: Deep multi-task and meta learning : Optimisation-based meta-learning. University Lecture, 2019. `http://cs330.stanford.edu/`.

[32] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Daan Wierstra, et al. Matching networks for one shot learning. In *Advances in neural information processing systems*, pages 3630–3638, 2016.

[33] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[34] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[35] Debajyoti Datta Ian Bunner Sebastien M.R. Arnold, Praateek Mahajan. learn2learn, September 2019.

[36] Luca Bertinetto, Joao F Henriques, Philip HS Torr, and Andrea Vedaldi. Meta-learning with differentiable closed-form solvers. *arXiv preprint arXiv:1805.08136*, 2018.

[37] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[38] Spyros Gidaris, Andrei Bursuc, Nikos Komodakis, Patrick Pérez, and Matthieu Cord. Boosting few-shot visual learning with self-supervision. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 8059–8068, 2019.

[39] Yonglong Tian, Yue Wang, Dilip Krishnan, Joshua B Tenenbaum, and Phillip Isola. Rethinking few-shot image classification: a good embedding is all you need? *arXiv preprint arXiv:2003.11539*, 2020.